HELLO WORLD

Welcome

Python is a programming language. Like other languages, it gives us a way to communicate ideas. In the case of a programming language, these ideas are "commands" that people use to communicate with a computer!

We convey our commands to the computer by writing them in a text file using a programming language. These files are called *programs*. Running a program means telling a computer to read the text file, translate it to the set of operations that it understands, and perform those actions.

Instructions

Change `Codecademy` to your name in the script to the right. Run the code to see what it does!

As soon as you're ready, move on to the next exercise to begin learning to write your own Python programs!

script.py

```
my_name = "Andruja"
print("Hello and welcome " + my_name + "!")
```

Comments

Ironically, the first thing we're going to do is show how to tell a computer to ignore a part of a program. Text written in a program but not run by the computer is called a comment. Python interprets anything after a `#` as a comment.

Comments can:

- Provide context for why something is written the way it is:

```
# This variable will be used to count the number of times
anyone tweets the word persnickety
persnickety_count = 0
```

- Help other people reading the code understand it faster:

```
# This code will calculate the likelihood that it will rain
tomorrow
complicated_rain_calculation_for_tomorrow()
```

- Ignore a line of code and see how a program will run without it:

```
# useful_value = old_sloppy_code()
useful_value = new_clean_code()
```

Instructions

1.

Documentation is an important step in programming. Write a comment describing the first program you want to write!

Hint

Start your comment with `#` and write the rest of your comment afterwards.

script.py

```
#This program calculates the addition of two numbers.
```

Print

Now what we're going to do is teach our computer to communicate. The gift of speech is valuable: a computer can answer many questions we have about "how" or "why" or "what" it is doing. In Python, the `print()` function is used to tell a computer to talk. The message to be printed should be surrounded by quotes:

```
# from Mary Shelley's Frankenstein
print("There is something at work in my soul, which I do not understand.")
```

In the above example, we direct our program to `print()` an excerpt from a notable book. The printed words that appear as a result of the `print()` function are referred to as *output*. The output of this example program would be:

```
There is something at work in my soul, which I do not understand.
```

Instructions

1.

Print the distinguished greeting "Hello world!"

Hint

It should look something like:

```
print("Hello world!")
```

script.py

```
print("Hello world!")
```

Strings

Computer programmers refer to blocks of text as strings. In our last exercise, we created the string “Hello world!”. In Python a string is either surrounded by double quotes (`"Hello world"`) or single quotes (`'Hello world'`). It doesn’t matter which kind you use, just be consistent.

Instructions

1.

Print your name using the `print()` command.

Hint

Use print syntax with either double-quotes or single-quotes to print your name, e.g.,

```
print("Codecademy")
```

2.

If your print statement uses double-quotes `"`, change them to single-quotes `'`. If it uses single-quotes `'`, change them to double-quotes `"`.

Try running your code again after switching the type of quote-marks. Is anything different about the output?

Hint

The output should stay the same!

script1.py

```
print("Andres R. Bucheli")
```

Variables

Programming languages offer a method of storing data for reuse. If there is a greeting we want to present, a date we need to reuse, or a user ID we need to remember we can create a variable which can store a value. In Python, we *assign* variables by using the equals sign (`=`).

```
message_string = "Hello there"
# Prints "Hello there"
print(message_string)
```

In the above example, we store the message “Hello there” in a variable called `message_string`. Variables can’t have spaces or symbols in their names other than an underscore (`_`). They can’t begin with numbers but they can have numbers after the first letter (e.g., `cool_variable_5` is OK).

It's no coincidence we call these creatures "variables". If the context of a program changes, we can update a variable but perform the same logical process on it.

```
# Greeting
message_string = "Hello there"
print(message_string)

# Farewell
message_string = "Hasta la vista"
print(message_string)
```

Above, we create the variable `message_string`, assign a welcome message, and print the greeting. After we greet the user, we want to wish them goodbye. We then update `message_string` to a departure message and print that out.

Instructions

1.
Update the variable `meal` to reflect each meal of the day before we print it.

script.py

```
# We've defined the variable "meal" here to the name of the food we ate for break
fast!
meal = "An english muffin"

# Printing out breakfast
print("Breakfast:")
print(meal)

# Now update meal to be lunch!
meal = "A sandwich"

# Printing out lunch
print("Lunch:")
print(meal)

# Now update "meal" to be dinner
meal = "Oatmeal"
# Printing out dinner
print("Dinner:")
print(meal)
```

Errors

Humans are prone to making mistakes. Humans are also typically in charge of creating computer programs. To compensate, programming languages attempt to understand and explain mistakes made in their programs.

Python refers to these mistakes as errors and will point to the location where an error occurred with a `^` character. When programs throw errors that we didn't expect to encounter we call those errors *bugs*. Programmers call the process of updating the program so that it no longer produces unexpected errors *debugging*.

Two common errors that we encounter while writing Python are `SyntaxError` and `NameError`.

- `SyntaxError` means there is something wrong with the way your program is written — punctuation that does not belong, a command where it is not expected, or a missing parenthesis can all trigger a `SyntaxError`.
- A `NameError` occurs when the Python interpreter sees a word it does not recognize. Code that contains something that looks like a variable but was never defined will throw a `NameError`.

Instructions

1.

You might encounter a `SyntaxError` if you open a string with a single quote and end it with double quotes. Update the string so that it starts and ends with the same punctuation.

You might encounter a `NameError` if you try to print a single word string but fail to put any quotes around it. Python expects the word of your string to be defined elsewhere but can't find where it's defined. Add quotes to either side of the string to squash this bug.

Update the malformed strings in the workspace to all be strings.

script.js

```
print('This message has mismatched quote marks!')
print('Abracadabra')
```

Numbers

Computers can understand much more than just strings of text. Python has a few numeric *data types*. It has multiple ways of storing numbers. Which one you use depends on your intended purpose for the number you are saving.

An *integer*, or `int`, is a whole number. It has no decimal point and contains all counting numbers (1, 2, 3, …) as well as their negative counterparts and the number 0. If you were counting the number of people in a room, the number of jellybeans in a jar, or the number of keys on a keyboard you would likely use an integer.

A *floating-point number*, or a `float`, is a decimal number. It can be used to represent fractional quantities as well as precise measurements. If you were measuring the length of your bedroom wall, calculating the average test score of a seventh-grade class, or storing a baseball player's batting average for the 1998 season you would likely use a `float`.

Numbers can be assigned to variables or used literally in a program:

```
an_int = 2
a_float = 2.1

print(an_int + 3)
# Output: 5
```

Above we defined an integer and a float as the variables `an_int` and `a_float`. We printed out the sum of the variable `an_int` with the number `3`. We call the number 3 here a *literal*, meaning it's actually the number `3` and not a variable with the number 3 assigned to it.

Floating-point numbers can behave in some unexpected ways due to how computers store them. For more information on floating-point numbers and Python, review Python's documentation on floating-point limitations.

Instructions

1.

A recent movie-going experience has you excited to publish a review. You rush out of the cinema and hastily begin programming to create your movie-review website: *The Big Screen's Greatest Scenes Decided By A Machine*.

Create the following variables and assign integer numbers to them: `release_year` and `runtime`.

Hint

Remember, integers are any whole numbers, positive or negative!
2.

Now, create the variable `rating_out_of_10` and assign it a float number between one and ten.

Hint

Remember, floats are numbers with a decimal point!

script.py

```
# Define the release and runtime integer variables below:
release_year = 2023;
runtime = 180;


# Define the rating_out_of_10 float variable below:
rating_out_of_10 = 9.5;
```

Calculations

Computers absolutely excel at performing calculations. The “compute” in their name comes from their historical association with providing answers to mathematical questions. Python performs the arithmetic operations of addition, subtraction, multiplication, and division with `+`, `-`, `*`, and `/`.

```
# Prints "500"
print(573 - 74 + 1)

# Prints "50"
print(25 * 2)

# Prints "2.0"
print(10 / 5)
```

Notice that when we perform division, the result has a decimal place. This is because Python converts all `int`s to `float`s before performing division. In older versions of Python (2.7 and earlier) this conversion did not happen, and integer division would always round down to the nearest integer.

Division can throw its own special error: `ZeroDivisionError`. Python will raise this error when attempting to divide by 0.

Mathematical operations in Python follow the standard mathematical order of operations.

Instructions

1.

Print out the result of this equation: `25 * 68 + 13 / 28`

Hint

You can just print a number without assigning it to a variable:

```
print(5 * 10)
```

This would print out `50`. Print out the given expression in a similar print statement.

script.py

```
print(25 * 68 + 13/28)
```

Changing Numbers

Variables that are assigned numeric values can be treated the same as the numbers themselves. Two variables can be added together, divided by `2`, and multiplied by a third variable without Python distinguishing between the variables and *literals* (like the number `2` in this example). Performing arithmetic on variables does not change the variable — you can only update a variable using the `=` sign.

```
coffee_price = 1.50
number_of_coffees = 4

# Prints "6.0"
print(coffee_price * number_of_coffees)
# Prints "1.5"
print(coffee_price)
# Prints "4"
print(number_of_coffees)

# Updating the price
coffee_price = 2.00

# Prints "8.0"
print(coffee_price * number_of_coffees)
# Prints "2.0"
print(coffee_price)
```

```python
# Prints "4"
print(number_of_coffees)
```

We create two variables and assign numeric values to them. Then we perform a calculation on them. This doesn't update the variables! When we update the `coffee_price` variable and perform the calculations again, they use the updated values for the variable!

Instructions

1.

You've decided to get into quilting! To calculate the number of squares you'll need for your first quilt let's create two variables: `quilt_width` and `quilt_length`. Let's make this first quilt 8 squares wide and 12 squares long.

2.

Print out the number of squares you'll need to create the quilt!

3.

It turns out that quilt required a little more material than you have on hand! Let's only make the quilt 8 squares long. How many squares will you need for this quilt instead?

script.py

```python
quilt_width = 8
quilt_length = 12

print(quilt_width * quilt_length)

quilt_length = 8

print(96 / quilt_length)
```

Exponents

Python can also perform exponentiation. In written math, you might see an exponent as a superscript number, but typing superscript numbers isn't always easy on modern keyboards. Since this operation is so related to multiplication, we use the notation `**`.

```python
# 2 to the 10th power, or 1024
print(2 ** 10)

# 8 squared, or 64
print(8 ** 2)
```

```
# 9 * 9 * 9, 9 cubed, or 729
print(9 ** 3)

# We can even perform fractional exponents
# 4 to the half power, or 2
print(4 ** 0.5)
```

Here, we compute some simple exponents. We calculate 2 to the 10th power, 8 to the 2nd power, 9 to the 3rd power, and 4 to the 0.5th power.

Instructions

1.

You really like how the square quilts from last exercise came out, and decide that all quilts that you make will be square from now on.

Using the exponent operator, print out how many squares you'll need for a 6x6 quilt, a 7x7 quilt, and an 8x8 quilt.

2.

Your 6x6 quilts have taken off so well, 6 people have each requested 6 quilts. Print out how many tiles you would need to make 6 quilts apiece for 6 people.

Hint

Since each person is requesting 6 quilts, we'll need `6 * 6 * 6 * 6` squares. Can you think of a way to express that as an exponent?

script.py

```
# Calculation of squares for:
# 6x6 quilt
print(6 ** 2)
# 7x7 quilt
print(7 ** 2)
# 8x8 quilt
print(8 ** 2)
# How many squares for 6 people to have 6 quilts each that are 6x6?
print(6 ** 4)
```

Modulo

Python offers a companion to the division operator called the modulo operator. The modulo operator is indicated by `%` and gives the remainder of a division calculation. If the number is divisible, then the result of the modulo operator will be 0.

```
# Prints 4 because 29 / 5 is 5 with a remainder of 4
print(29 % 5)

# Prints 2 because 32 / 3 is 10 with a remainder of 2
print(32 % 3)

# Modulo by 2 returns 0 for even numbers and 1 for odd numbers
# Prints 0
print(44 % 2)
```

Here, we use the modulo operator to find the remainder of division operations. We see that `29 % 5` equals 4, `32 % 3` equals 2, and `44 % 2` equals 0.

The modulo operator is useful in programming when we want to perform an action every nth-time the code is run. Can the result of a modulo operation be larger than the divisor? Why or why not?

Instructions

1.

You're trying to divide a group into four teams. All of you count off, and you get number 27.

Find out your team by computing 27 modulo 4. Save the value to `my_team`.

2.

Print out `my_team`. What number team are you on?

3.

Food for thought: what number team are the two people next to you (26 and 28) on? What are the numbers for all 4 teams? (Optional Challenge Question)

Hint

To see the modulo values of those near you, use:

```
print(26 % 4)
print(28 % 4)
```

script.py

```
my_team = 27 % 4
print(my_team)
print(26 % 4)
print(28 % 4)
```

Concatenation

The `+` operator doesn’t just add two numbers, it can also “add” two strings! The process of combining two strings is called *string concatenation*. Performing string concatenation creates a brand new string comprised of the first string’s contents followed by the second string’s contents (without any added space in-between).

```
greeting_text = "Hey there!"
question_text = "How are you doing?"
full_text = greeting_text + question_text

# Prints "Hey there!How are you doing?"
print(full_text)
```

In this sample of code, we create two variables that hold strings and then concatenate them. But we notice that the result was missing a space between the two, let’s add the space in-between using the same concatenation operator!

```
full_text = greeting_text + " " + question_text

# Prints "Hey there! How are you doing?"
print(full_text)
```

Now the code prints the message we expected.

If you want to concatenate a string with a number you will need to make the number a string first, using the `str()` function. If you’re trying to `print()` a numeric variable you can use commas to pass it as a different argument rather than converting it to a string.

```
birthday_string = "I am "
age = 10
birthday_string_2 = " years old today!"

# Concatenating an integer with strings is possible if we turn the
integer into a string first
full_birthday_string = birthday_string + str(age) + birthday_string_2

# Prints "I am 10 years old today!"
print(full_birthday_string)
```

```
# If we just want to print an integer
# we can pass a variable as an argument to
# print() regardless of whether
# it is a string.

# This also prints "I am 10 years old today!"
print(birthday_string, age, birthday_string_2)
```

Using `str()` we can convert variables that are not strings to strings and then concatenate them. But we don't need to convert a number to a string for it to be an argument to a print statement.

Instructions

1.

Concatenate the strings and save the message they form in the variable `message`.

Now uncomment the print statement and run your code to see the result in the terminal!

script.py

```
string1 = "The wind, "
string2 = "which had hitherto carried us along with amazing rapidity, "
string3 = "sank at sunset to a light breeze; "
string4 = "the soft air just ruffled the water and "
string5 = "caused a pleasant motion among the trees as we approached the shore, "
string6 = "from which it wafted the most delightful scent of flowers and hay."

# Define message below:
message = string1 + string2 + string3 + string4 + string5 + string6


print(message)
```

Plus Equals

Python offers a shorthand for updating variables. When you have a number saved in a variable and want to add to the current value of the variable, you can use the `+=` (plus-equals) operator.

```
# First we have a variable with a number saved
number_of_miles_hiked = 12
```

```
# Then we need to update that variable
# Let's say we hike another two miles today
number_of_miles_hiked += 2

# The new value is the old value
# Plus the number after the plus-equals
print(number_of_miles_hiked)
# Prints 14
```

Above, we keep a running count of the number of miles a person has gone hiking over time. Instead of recalculating from the start, we keep a grand total and update it when we've gone hiking further.

The plus-equals operator also can be used for string concatenation, like so:

```
hike_caption = "What an amazing time to walk through nature!"

# Almost forgot the hashtags!
hike_caption += " #nofilter"
hike_caption += " #blessed"
```

We create the social media caption for the photograph of nature we took on our hike, but then update the caption to include important social media tags we almost forgot.

Instructions

1.

We're doing a little bit of online shopping and find a pair of new sneakers. Right before we check out, we spot a nice sweater and some fun books we also want to purchase!

Use the `+=` operator to update the `total_price` to include the prices of `nice_sweater` and `fun_books`.

The prices (also included in the workspace) are:

- `new_sneakers = 50.00`
- `nice_sweater = 39.00`
- `fun_books = 20.00`

script.py

```
total_price = 0


new_sneakers = 50.00
```

```
total_price += new_sneakers

nice_sweater = 39.00
fun_books = 20.00
# Update total_price here:
total_price += nice_sweater
total_price += fun_books

print("The total price is", total_price)
```

Multi-line Strings

Python strings are very flexible, but if we try to create a string that occupies multiple lines we find ourselves face-to-face with a `SyntaxError`. Python offers a solution: *multi-line* strings. By using three quote-marks (`"""` or `'''`) instead of one, we tell the program that the string doesn't end until the next triple-quote. This method is useful if the string being defined contains a lot of quotation marks and we want to be sure we don't close it prematurely.

```
leaves_of_grass = """
Poets to come! orators, singers, musicians to come!
Not to-day is to justify me and answer what I am for,
But you, a new brood, native, athletic, continental, greater than
  before known,
Arouse! for you must justify me.
"""
```

In the above example, we assign a famous poet's words to a variable. Even though the quote contains multiple linebreaks, the code works!

If a multi-line string isn't assigned a variable or used in an expression it is treated as a comment.

Instructions

1.

Assign the string

```
Stranger, if you passing meet me and desire to speak to me, why
  should you not speak to me?
And why should I not speak to you?
```

to the variable `to_you`.

script.py

```
# Assign the string here
to_you = """
  Stranger, if you passing meet me and desire to speak to me, why
  should you not speak to me?
And why should I not speak to you?
"""


print(to_you)
```

Review

In this lesson, we accomplished a lot of things! We instructed our computers to print messages, we stored these messages as variables, and we learned to update those messages depending on the part of the program we were in. We performed mathematical calculations and explored some of the mathematical expressions that Python offers us. We learned about errors and other valuable skills that will continue to serve us as we develop our programming skills.

Good job!

Here are a few more resources to add to your toolkit:

- Codecademy Docs: Python
- Codecademy Workspaces: Python

Make sure to bookmark these links so you have them at your disposal.

Instructions

1.

Create variables:

- `my_age`
- `half_my_age`
- `greeting`
- `name`
- `greeting_with_name`

Assign values to each using your knowledge of division and concatenation!

script.py

```
my_age = 45
half_my_age = 45 / 2
greeting = 'Hello'
name = 'Andres R. Bucheli'
greeting_with_name = greeting + ' ' + name

print(greeting_with_name)
```